CHAPTER 1

ASP .NET Core MVC in Context

ASP.NET Core MVC is a web application development framework from Microsoft that combines the effectiveness and tidiness of model-view-controller (MVC) architecture, ideas and techniques from agile development, and the best parts of the .NET platform. In this chapter, you'll learn why Microsoft created ASP.NET Core MVC, see how it compares to its predecessors and alternatives, and, finally, get an overview of what's new in ASP.NET Core MVC and what's covered in this book.

Understanding the History of ASP.NET Core MVC

The original ASP.NET was introduced in 2002, at a time when Microsoft was keen to protect a dominant position in traditional desktop application development and saw the Internet as a threat. Figure 1-1 illustrates Microsoft's technology stack as it appeared then.

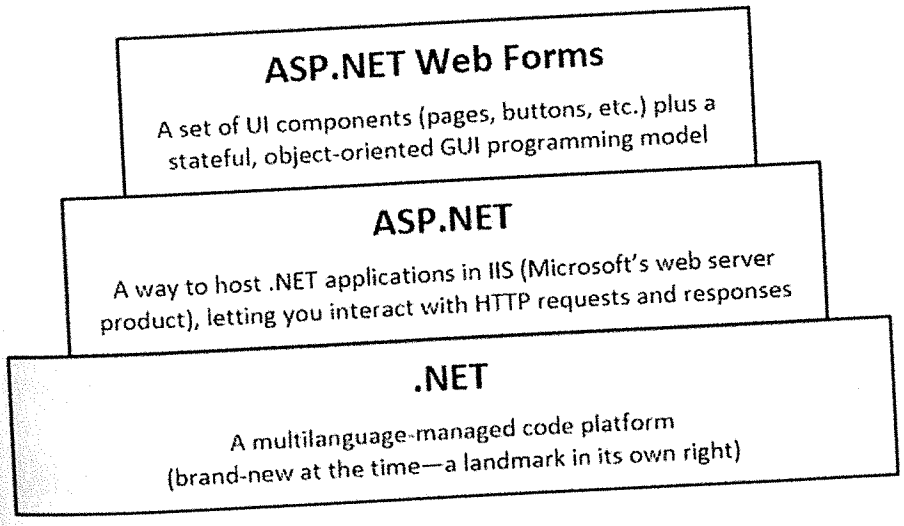

Figure 1-1. *The ASP.NET Web Forms technology stack*

A. Freeman, *Pro ASP.NET Core MVC 2*, https://doi.org/10.1007/978-1-4842-3150-0_1

ASP.NET Web Forms

With Web Forms, Microsoft attempted to hide both Hypertext Transfer Protocol (HTTP), with its intrinsic statelessness, and Hypertext Markup Language (HTML), which at the time was unfamiliar to many developers, by modeling the user interface (UI) as a hierarchy of server-side control objects. Each control kept track of its own state across requests, rendering itself as HTML when needed and automatically connecting client-side events (for example, a button click) with the corresponding server-side event handler code. In effect, Web Forms is a giant abstraction layer designed to deliver a classic event-driven graphical user interface (GUI) over the Web.

The idea was to make web development feel just the same as developing a desktop application. Developers could think in terms of a stateful UI and didn't need to work with a series of independent HTTP requests and responses. Microsoft could seamlessly transition the army of Windows desktop developers into the new world of web applications.

What Was Wrong with ASP.NET Web Forms?

Traditional ASP.NET Web Forms development was good in principle, but reality proved more complicated.

- *View State weight*: The actual mechanism for maintaining state across requests (known as View State) resulted in large blocks of data being transferred between the client and server. This data could reach hundreds of kilobytes in even modest web applications, and it went back and forth with every request, leading to slower response times and increasing the bandwidth demands of the server.
- *Page life cycle*: The mechanism for connecting client-side events with server-side event handler code, part of the page life cycle, could be complicated and delicate. Few developers had success manipulating the control hierarchy at runtime without creating View State errors or finding that some event handlers mysteriously failed to execute.
- *False sense of separation of concerns*: ASP.NET Web Forms' *code-behind* model provided a means to take application code out of its HTML markup and into a separate code-behind class. This was done to separate logic and presentation, but, in reality, developers were encouraged to mix presentation code (for example, manipulating the server-side control tree) with their application logic (for example, manipulating database data) in these same monstrous code-behind classes. The end result could be fragile and unintelligible.
- *Limited control over HTML*: Server controls rendered themselves as HTML, but not necessarily the HTML you wanted. In early versions of ASP.NET, the HTML output failed to meet web standards or make good use of Cascading Style Sheets (CSS), and server controls generated unpredictable and complex ID attributes that were hard to access using JavaScript. These problems have improved in recent Web Forms releases, but it can still be tricky to get the HTML you expect.
- *Leaky abstraction*: Web Forms tried to hide HTML and HTTP wherever possible. As you tried to implement custom behaviors, you frequently fell out of the abstraction, which forced you to reverse-engineer the postback event mechanism or perform obtuse acts to make it generate the desired HTML.
- *Low testability*: The designers of Web Forms could not have anticipated that automated testing would become an essential component of software development. The tightly coupled architecture they designed was unsuitable for unit testing. Integration testing could be a challenge, too.

Web Forms wasn't all bad, and eventually, Microsoft put effort into improving standards compliance and simplifying the development process and even took some features from the original ASP.NET MVC Framework and applied them to Web Forms. Web Forms excelled when you needed quick results, and you could have a reasonably complex web app up and running within a day. But unless you were careful during development, you would find that the application you created was hard to test and hard to maintain.

The Original MVC Framework

In October 2007, Microsoft announced a new development platform, built on the existing ASP.NET platform, that was intended as a direct response to the criticisms of Web Forms and the popularity of competing platforms such as Ruby on Rails. The new platform was called the ASP.NET MVC Framework and reflected the emerging trends in web application development, such as HTML and CSS standardization, RESTful web services, effective unit testing, and the idea that developers should embrace the stateless nature of HTTP.

The concepts that underpin the original MVC Framework seem natural and obvious now, but they were lacking from the world of .NET web development in 2007. The introduction of the ASP.NET MVC Framework brought Microsoft's web development platform back into the modern age.

The MVC Framework also signaled an important change in attitude from Microsoft, which had previously tried to control every component in the web application toolchain. With the MVC Framework, Microsoft built on open source tools such as jQuery, took on design conventions and best practices from competing (and more successful) platforms, and released the source code to the MVC Framework for developers to inspect.

What Was Wrong with the Original MVC Framework?

At the time the MVC Framework was created, it made sense for Microsoft to create it on top of the existing ASP.NET platform, which had a lot of solid low-level functionality that provided a head start in the development process and which was already well-known and understood by ASP.NET developers.

But compromises were required to graft the MVC Framework onto a platform that was originally designed for Web Forms. MVC Framework developers became used to using configuration settings and code tweaks that disabled or reconfigured features that didn't have any bearing on their web application but were required to get everything working.

As the MVC Framework grew in popularity, Microsoft started to take some of the core features and add them to Web Forms. The result was increasingly odd, where features with design quirks required to support the MVC Framework were extended to support Web Forms, with further design quirks to make everything fit together. At the same time, Microsoft started to expand ASP.NET with new frameworks for creating web services (Web API) and real-time communication (SignalR). The new frameworks added their own configuration and development conventions, each of which had its own benefits and oddities, and the overall result was a fragmented mess.

Understanding ASP.NET Core

In 2015, Microsoft announced a new direction for ASP.NET and the MVC Framework, which would eventually produce ASP.NET Core MVC, the topic of this book.

ASP.NET Core is built on .NET Core, which is a cross-platform version of the .NET Framework without the Windows-specific application programming interfaces (APIs). Windows is still a dominant operating system, but web applications are increasingly hosted in small and simple containers in cloud platforms, and by embracing a cross-platform approach, Microsoft has extended the reach of .NET, made it possible to deploy ASP.NET Core applications to a broader set of hosting environments, and, as a bonus, made it possible for developers to create ASP.NET Core web applications on Linux and macOS.

ASP.NET Core is a completely new framework. It is simpler, it is easier to work with, and it is free of the legacy that comes from Web Forms. And, since it is based on .NET Core, it supports the development of web applications on a range of platforms and containers.

ASP.NET Core MVC provides the functionality of the original ASP.NET MVC Framework built on the new ASP.NET Core platform. It integrates the features that were previously provided by Web API, it includes a more natural way of generating complex content, and it makes key development tasks, such as unit testing, simpler and more predictable.

What's New in ASP.NET Core MVC 2

The ASP.NET Core MVC 2 release focuses on consolidation, working through some of the tooling and platform changes that were introduced in earlier versions. ASP.NET Core MVC 2 requires .NET Core 2, which has a much-expanded API surface and is now supported on additional Linux distributions. Useful changes include a new meta-package system, which simplifies the management of NuGet packages, a new configuration system for ASP.NET Core, and support for Entity Framework Core 2. The biggest new feature is Razor Pages, which is an attempt to re-create the development style associated with Web Pages using a more modern platform, but Razor Pages will not be of interest for MVC developers (and I do not describe it in this book).

Key Benefits of ASP.NET Core MVC

The following sections briefly describe how the new MVC platform overcomes the legacy of Web Forms and the original MVC Framework and how it has brought ASP.NET back to the cutting edge.

MVC Architecture

ASP.NET Core MVC follows a pattern called *model-view-controller*, which guides the shape of an ASP.NET web application and the interactions between the components it contains.

It is important to distinguish between the MVC architectural pattern and the ASP.NET Core MVC implementation. The MVC pattern is not new—it dates back to 1978 and the Smalltalk project at Xerox PARC—but it has gained popularity today as a pattern for web applications for the following reasons:

- User interaction with an application that adheres to the MVC pattern follows a natural cycle: the user takes an action, and in response, the application changes its data model and delivers an updated view to the user. Then the cycle repeats. This is a convenient fit for web applications delivered as a series of HTTP requests and responses.
- Web applications necessitate combining several technologies (databases, HTML, and executable code, for example), usually split into a set of tiers or layers. The patterns that arise from these combinations map naturally onto the concepts in the MVC pattern.

ASP.NET Core MVC implements the MVC pattern and, in doing so, provides a greatly improved separation of concerns when compared to Web Forms. In fact, ASP.NET Core MVC implements a variant of the MVC pattern that is especially suitable for web applications. You will learn more about the theory and practice of this architecture in Chapter 3.

Extensibility

ASP.NET Core and ASP.NET Core MVC are built as a series of independent components that have well-defined characteristics, satisfy a .NET interface, or are built on an abstract base class. You can easily replace key components with ones of your own implementation. In general, ASP.NET Core MVC gives you these three options for each component:

- Use the *default* implementation of the component as it stands (which should be enough for most applications).
- Derive a *subclass* of the default implementation to tweak its behavior.
- *Replace* the component entirely with a new implementation of the interface or abstract base class.

You'll learn all about the various components and how and why you might want to tweak or replace each of them, starting in Chapter 14.

Tight Control over HTML and HTTP

ASP.NET Core MVC produces clean, standards-compliant markup. Its built-in tag helpers produce standards-compliant output, but there is a more significant philosophical change compared with Web Forms. Instead of generating out swathes of HTML over which you have little control, ASP.NET Core MVC encourages you to craft simple, elegant markup styled with CSS.

Of course, if you do want to throw in some ready-made widgets for complex UI elements such as date pickers or cascading menus, the "no special requirements" approach taken by ASP.NET Core MVC makes it easy to use best-of-breed client-side libraries such as jQuery, Angular, React, or the Bootstrap CSS library. ASP.NET Core MVC meshes so well with these libraries that Microsoft includes templates that incorporate them to jump-start new development projects.

ASP.NET Core MVC works in tune with HTTP. You have control over the requests passing between the browser and server, so you can fine-tune your user experience as much as you like. Ajax is made easy, and creating web services to receive browser HTTP requests is a simple process, as described in Chapter 20.

Testability

The ASP.NET Core MVC architecture gives you a great start in making your application maintainable and testable because you naturally separate different application concerns into independent pieces. In addition, each piece of the ASP.NET Core platform and the ASP.NET Core MVC framework can be isolated and replaced for unit testing, which can be performed using any popular open source testing framework, such as xUnit, which I introduce in Chapter 7.

In this book, you will see examples of how to write clean, simple unit tests for ASP.NET MVC controllers and actions that supply fake or mock implementations of framework components to simulate any scenario, using a variety of testing and mocking strategies. Even if you have never written a unit test before, you will be off to a great start.

Testability is not only a matter of unit testing. ASP.NET Core MVC applications work well with UI automation testing tools, too. You can write test scripts that simulate user interactions without needing to guess which HTML element structures, CSS classes, or IDs the framework will generate, and you do not have to worry about the structure changing unexpectedly.

Powerful Routing System

The style of uniform resource locators (URLs) has evolved as web application technology has improved. URLs like this one:

```
/App_v2/User/Page.aspx?action=show%20prop&prop_id=82742
```

are increasingly rare, replaced by a simpler, cleaner format like this:

```
/to-rent/chicago/2303-silver-street
```

There are some good reasons for caring about the structure of URLs. First, search engines give weight to keywords found in a URL. A search for "rent in Chicago" is much more likely to turn up the simpler URL. Second, many web users are now savvy enough to understand a URL and appreciate the option of navigating by typing it into their browser's address bar. Third, when someone understands the structure of a URL, they are more likely to link to it, share it with a friend, or even read it aloud over the phone. Fourth, it doesn't expose the technical details, folder, and file name structure of your application to the public Internet, so you are free to change the underlying implementation without breaking all your incoming links.

Clean URLs were hard to implement in earlier frameworks, but ASP.NET Core MVC uses a feature known as *URL routing* to provide clean URLs by default. This gives you control over your URL schema and its relationship to your application, offering you the freedom to create a pattern of URLs that is meaningful and useful to your users, without the need to conform to a predefined pattern. And, of course, this means you can easily define a modern REST-style URL schema if you want. You'll find a thorough description of URL routing in Chapters 15 and 16.

Modern API

Microsoft's .NET platform has evolved with each major release, supporting—and even defining—the state-of-the-art aspects of modern programming. ASP.NET Core MVC is built for .NET Core, so its API can take full advantage of language and runtime innovations familiar to C# programmers, including the `await` keyword, extension methods, lambda expressions, anonymous and dynamic types, and Language Integrated Query (LINQ).

Many of the ASP.NET Core MVC API methods and coding patterns follow a cleaner, more expressive composition than was possible with earlier platforms. Don't worry if you are not up to speed with the latest C# language features; I provide a summary of the most important C# features for MVC development in Chapter 4.

Cross-Platform

Previous versions of ASP.NET were specific to Windows, requiring a Windows desktop to write web applications and a Windows server to deploy and run them. Microsoft made ASP.NET Core cross-platform, both for development and for deployment. .NET Core is available for different platforms, including macOS and a range of popular Linux distributions. Cross-platform support makes it easier to deploy ASP.NET Core MVC applications, and there is good support for working with application container platforms, such as Docker.

Most ASP.NET Core MVC development is likely to be done using Visual Studio for the immediate future, but Microsoft has also created a cross-platform development tool called Visual Studio Code, which means that ASP.NET Core MVC development is no longer restricted to Windows.

ASP.NET Core MVC Is Open Source

Unlike previous Microsoft web development platforms, you are free to download the source code for ASP.NET Core and ASP.NET Core MVC and even modify and compile your own version of it. This is invaluable when your debugging trail leads into a system component and you want to step into its code (and even read the original programmer's comments). It is also useful if you are building an advanced component and want to see what development possibilities exist or how the built-in components actually work.

You can download the ASP.NET Core and ASP.NET Core MVC source code from `https://github.com/aspnet`.

What Do I Need to Know?

To get the most from this book, you should be familiar with the basics of web development, understand how HTML and CSS work, and have a working knowledge of C#. Don't worry if you are a little hazy on the client-side details, such as JavaScript. My emphasis is on server-side development in this book, and you can pick up what you need through the examples. In Chapter 4, I summarize the most useful C# language features for MVC development, which you'll find useful if you are moving to the latest .NET versions from an earlier release.

What Is the Structure of This Book?

This book is split into two parts, each of which covers a set of related topics.

Part 1: Introducing ASP.NET Core MVC

I start this book by putting ASP.NET Core MVC in context. I explain the benefits and practical impact of the MVC pattern, cover the way in which ASP.NET Core MVC fits into modern web development, and describe the tools and C# language features that every ASP.NET Core MVC programmer needs.

In Chapter 2, you will dive right in by creating a simple web application and will get an idea of what the major components and building blocks are and how they fit together. Most of this part of the book, however, is given over to the development of a project called SportsStore, through which I show you a realistic development process from inception to deployment, touching on the major features of ASP.NET Core MVC.

Part 2: ASP.NET Core MVC in Detail

In Part 2, I explain the inner workings of ASP.NET Core MVC features that I used to build the SportsStore application. I show you how each feature works, explain the role it plays, and show you the configuration and customization options that are available. Having set the broad context in Part 1, I dig right into the details in Part 2.

Where Can You Get the Example Code?

You can download the example projects for all the chapters in this book from `https://github.com/apress/pro-asp.net-core-mvc-2`. The download is available without charge and includes all the supporting resources that are required to re-create the examples without having to type them in. You don't have to download the code, but it is the easiest way of experimenting with the examples and makes it easy to copy and paste code into your own projects.

Where Can You Get Corrections for This Book?

You can find corrections for this book in the Errata file in the GitHub repository for this book (`https://github.com/apress/pro-asp.net-core-mvc-2`).

Contacting the Author

If you have problems making the examples in this chapter work or if you find a problem in the book, then you can e-mail me at `adam@adam-freeman.com`, and I will try my best to help. Please check the errata for this book at `https://github.com/apress/pro-asp.net-core-mvc-2` to see if it contains a solution to your problem before contacting me.

Summary

In this chapter, I explained the context in which ASP.NET Core MVC exists and how it has evolved from Web Forms and the original ASP.NET MVC Framework. I described the benefits of using the ASP.NET Core MVC and the structure of this book. In the next chapter, you'll see ASP.NET Core MVC in action in a simple demonstration of the features that deliver these benefits.